**Python (version ≥3.6) for R Users: Fundamentals**

**CMU MSP 36601, Fall 2017, Howard Seltman**

1. ***Why learn/use Python?***
   1. Python is increasing listed as a job skill for data scientists.
   2. Python implements object oriented programming more fully than R.
   3. Python has a small core language and a huge set of modules to extend it. Python tends to be written with explicit scope syntax, e.g., myModule.myFunction(), to make this obvious.
   4. Python is used more than R when the data analysis tasks need to be integrated with web apps or if statistics code needs to be incorporated into a production database.
   5. Python is less quirky than R from a computer science point of view.
   6. Python code enforces some degree of readability.
   7. Cython works well to compile code for increased speed (requires separate C compiler).
2. The recommended **Integrated Development Environment,** “Spyder”, is loaded as part of “Anaconda” (at <https://www.anaconda.com/download/>) along with Python and many useful modules. It works well on Windows, Linux, and Mac. Be sure to choose Python 3.x when downloading. The current version of Spyder has an “IPython” console that supports a full feature editor with syntax coloring, code validation, style checking, code completion, and “go to definition”; integrated documentation; integrated debugging; and some project capabilities.

Once you install Anaconda, do the following immediately, and every few weeks:

On a PC, use Start / All Programs / Accessories / right-click on Command Prompt / Run as administrator. On a Mac use Launchpad / Other / Terminal.

Then, at the prompt enter each of these (one by one), answer “y” to the questions:

1) conda update qt qtpy

2) conda update spyder

3) conda update python

4) exit

Python 2 is not acceptable for this class. It is deprecated and will no longer be supported as of next year. Running Python 3 in other than Spyder is OK, but I may not be able to help you if you have a problem.

Spyder has a nice, annoying feature that helps produce perfectly styled Python code: Go to the menu item python / Preferences /Editor / Code Introspection & Analysis, and check “Real-type code style analysis”. Do it! Now! I will complain when you violate “official” Python style. But if you respond to the little orange triangles with exclamation marks, you will quickly learn to write with perfect style.

1. ***Quick Start: A data simulation and analysis***

# This is quickStart.py

# Declare extensions to the core language:

import numpy as np

import pandas as pd

import random

import statsmodels.formula.api as smf

# Create subject characteristics for simulated data

N = 20 # N <= 26

# First create lists as "list comprehensions"

id = [chr(ord('A') + ii) for ii in range(N)]

ages = [random.choice(list(range(20, 35))) for ii in range(N)]

male = [0 if random.random() < 0.5 else 1 for ii in range(N)]

data = pd.DataFrame({"id": id, "ages": ages, "male": male},

index=['P' + str(ii) for ii in range(N)])

print(data)

print(type(data))

print(type(data["ages"])) # columns are of class "Series"

# Generate simulated regression data

b = (3.0, 2.5, -1.4)

k = len(b) - 1

sig = 2.5

X = [[1] + [random.normalvariate(0, 1) for x in range(k)]

for y in range(N)]

y = [random.normalvariate(np.dot(x, b), sig) for x in X]

reg = np.array([ivs[1:] + [dv] for ivs, dv in zip(X, y)])

colnames = ["x" + str(1 + ii) for ii in range(k)] + ["y"]

# Create a DataFrame from a numpy array

reg = pd.DataFrame(reg, columns=colnames)

# (inefficient) matrix approach to regression

Xa = np.array(X)

bhat = np.linalg.inv(Xa.transpose() @ X) @ Xa.transpose() @ y

print(bhat)

# Regression with module 'statsmodels'

m0 = smf.ols('y ~ x1 + x2', data=reg, hasconst=True).fit()

print(m0.summary())

1. Be sure to use “Python 3” in Google ***searches***.
2. Some ***key characterization of Python***
   1. Python is an **interpreted** language (so relatively slow, like R).
   2. Because Guido van Rossum, inventor and “benevolent dictator for life” is a fan of [Monty Python's Flying Circus](https://en.wikipedia.org/wiki/Monty_Python%27s_Flying_Circus), the documentation tends to use “**spam**” and “**eggs**” instead of “foo” and “bar”. You may want to view a [sample video clip](https://www.youtube.com/watch?v=vfHNsbHcFLs&list=PLE1A49EE04D953709).
   3. The core language is fairly small, and many **optional modules** extend basic Python, e.g., “1/10+1/10+1/10==3/10” is False, as in R, but the “fractions” module can give the correct answer. Very little useful statistical programming can be done without additional modules.
   4. Unlike R, Python is ***not inherently vector based***, e.g., “+” for lists does concatenation.
   5. Standard ***arithmetic operator***s are used, plus “\*\*” for “to the power of”, “%” for modulo (“%%” in R) and “//” for integer divide (“%/%” in R).
   6. Like R, ***assignment*** uses an equal sign (not that ugly “<-“ pair ☺) and ***equality testing*** uses “==” and “!=”. To ***delete variables,*** instead of R’s rm() function, use the del statement (e.g., del x).
   7. ***Octal, hexadecimal, and bitwise constants*** are built in and use a leading zero. E.g., 0o12, 0xA, and 0b1010 are all ways to write decimal 10 (but 010 is illegal!).
   8. Standard **logical operators** are used, but they are combined with “and”, “or”, and “not”. Similar to SAS, and different from R, if x=3, then 1 < x < 5 is True and 5 < x < 7 is False.
   9. Python uses “|”, “&”, and “^” for ***bitwise*** “or”, “and” and “exclusive or”, and “<<” and “>>” for ***bit shifting***. To demonstrate, we introduce the bin() function which converts an integer to base 2 (as a string).

>>> bin(0b101 << 1)

'0b1010'

>>> bin(0b100 >> 1)

'0b10'

>>> bin(0b1001 | 0b1010)

'0b1011'

>>> bin(0b1001 & 0b1010)

'0b1000'

>>> bin(0b1001 ^ 0b1010)

'0b11'

* 1. Python uses “#” for ***comments***.
  2. ***Function arguments*** must be placed so that the *positional* ones are first, followed by *fully named* ones (if you are overriding the default).
  3. Python is flexible in ***spacing*** between elements of a code line.
  4. Like R, Python allows ***multiple code lines on one physical line*** using “;”.
  5. Code lines must be i) syntactically complete; ii) or break inside the paired elements [], (), or {}; iii) or use a terminal backslash to indicate ***line continuation***.
  6. Unusually, Python requires the use of ***indenting to indicate code blocks*** (vs. R’s {}).
  7. Python has an official ***style guide*** in Python Enhancement Protocol [PEP 8](https://www.python.org/dev/peps/pep-0008/). See <https://www.python.org/dev/peps/> for other PEPs.

1. Python ***variable names*** are case sensitive, may include underscores (but not periods), and may be long. Python style prefers underscores between words rather than camel case for ordinary variables and function names, and UpperCamelCase for class names.
2. Python supports **multiple assignment** statements:

>>> t = 2, 3, 7 # not multiple; more clear as: t = (2, 3, 7)

>>> t

(2, 3, 7)

>>> a, b = 1, 5 # or (a, b) = (1, 5) or a, b = [1, 5]

>>> b

5

>>> a, b = b, a

>>> b

1

1. Python has **immutable** and **mutable** basic ***data types*,** based on whether the data can be changed “in place”. Some other classifications include “sequences”, “sets”, and “mappings”. In addition, many “container” objects that can hold more than one value support ***indexing*** to “pull out” specific values. ***Indexing is zero based.*** Many objects that hold more than one value support **iteration**, which basically means that they can be the control of a “for” loop. The ***type() function*** can be used to show the type (equivalent to class() in R). Unlike R, non-sequences (i.e., “atomic” variables) have no length and cannot be indexed.
   1. ***Special type*: “**None” is like R’s “NULL”.
   2. **Immutable non-sequences’s** hold one thing
      1. A ***boolean*** (“bool”) is either “True” or “False” (result of any comparison)
      2. An ***integer*** (“int”) number is not restricted to -2147483647 to +2167483847 as in R and most other languages
      3. IEEE ***floating point*** “float” numbers are double precision and are restricted to roughly 15 significant figures and less than 10308. Number smaller than about 10-324 are indistinguishable from zero.
      4. ***Complex numbers*** (“complex”) use “j” (not “i"), e.g., (3+2j) \* (3-2j) evaluates to (13+0j).
   3. **Immutable sequences** can have length greater than one, but cannot be changed “in place”. Sequences, whether immutable or mutable, support the “in” and “not in” syntax to check if a single value is in the sequence. Two sequences can be concatenated using the “+” operator and concatenated with multiple copies of themselves with “\*” followed by an integer, e.g., (1, 2)\*3 evaluates to (1, 2, 1, 2, 1, 2).
      1. ***Strings*** (“str”) hold Unicode text (in UTF-8 multi-byte format). Single and double quoted strings allow the other inside. Triple quoted strings can cover multiple lines with using a “\” to indicate line continuation. The form "x\\ty" has 4 characters, the second of which is a backslash, while "x\ty" has 3 characters, the second of which is a tab. “Raw strings” start with “r” or “R” and do not give backslash special meaning, so r"x\ty" has length 4. Inside a string, a “\u” (or “\U”) prefix denotes a Unicode character in 4 (or 8) hexadecimal digits, so "alpha=\u03b1" evaluates to “alpha=”. The \N{} syntax allows named characters, so "alpha=\N{greek small letter alpha}" evaluates to “alpha=”. (See <https://unicode-table.com/en/#ipa-extensions> and <http://www.charbase.com/block/greek-and-coptic>.) If we define x = "ABC", then len(x) is 3 and x[0] equals “A”, but the assignment x[0] = "B" is invalid because string objects are immutable.
      2. ***Bytes*** (“bytes”) are sequences of integers (0-255) that represent ASCII values as single bytes using a “b” or “B” prefix. For example if we define x = b"ABC", then len(x) is 3 and x[0] equals 65, but the assignment x[0] = 66 is invalid because byte objects are immutable.
      3. A **tuple** is an immutable, indexable sequence of any length, possibly of mixed type. Tuples are indicated by commas, e.g., x = (3, "ABC", 2.5-0j) is a tuple where type(x[2]) is “complex”. The parentheses are optional, but always shown when examining tuples. Because of possible confusion with ordinary numbers, a ***tuple of length one*** must use a trailing comma, e.g., x = (3,) is a tuple while x = (3) is an ordinary (non-indexable) integer. Tuples are commonly used as the return value of functions that return multiple values. Tuples are more efficient than lists (see below) when the “vector” will not change over the whole program.
      4. A **frozenset** is an immutable version of a set (see below).
   4. ***Mutable non-sequence***

A **dictionary** (“dict”) is the only standard mappable type. It is indexed by unique “hashable keys”, which return arbitrary objects (may be mixed types). The key can be numeric, string, and several other types. *There is no definite order to the objects in a dictionary*. Dictionary constants are enclosed in curly braces. An empty dictionary can be created using x = {}. A dictionary that returns the numbers for number words can be defined with num\_words = {"one": 1, "two": 2, "four": 4, "three": 3}. Because there is no order, num\_words[0] is invalid. But num\_words["four"] returns the value 4. We can then use num\_words["six"] = 6 to add an element. And if we redefine an element, e.g., num\_words["four"] = 44, then the old value of 4 is overwritten with the new value of 44.

* 1. ***Mutable sequences*** 
     1. A **list** is a mutable, indexed sequence of possibly mixed type. (A tuple is better if no changes to the values are expected.) Lists are the workhorses of Python. Lists may be created using square bracket notation, e.g. x = [1, 2.4, "MSP"), or using the list() function on some other data types. The “in” and “not in” operators as well as “+” for concatenation and “\*” for repetition are supported. It is very similar to an R list, and is used more often in Python that lists are in R.
     2. A **byte array** is a mutable version of a byte object. It can be created with bytearray(b"ABC") and is not used much.
  2. A **range** is an object that supports iteration (e.g., in a “for” loop) without growing in size when it refers to a larger number of values. The syntax covers examples like range(50) which iterates over 0, 1, …, 48, 49 (50 different values), range(5, 7) which iterates over 5 and 6, and range(12, 20, 2) which iterates over 12, 14, 16, 18.
  3. A **set** is a mutable built-in fundamental type that is not indexable and that cannot contain any value more than once. Sets are iterable. They may be of mixed types (but usually not). Sets may be created by using curly braces (without colons) or with the set() function using a list or tuple as the argument. E.g., if we define x = {1, 2, 2, 1, 3, 1} we get an object of length 3, which when examined shows as {1, 2, 3}. This is similar to x = unique(c(1, 2, 1, 3, 1)) in R.
  4. ***Strange and dangerous copy behavior of Python***

Unlike R, Python allows two names to refer to the same data. Think of this as Python trying to save space and time by not making copies of data. This can be quite confusing and can cause hard-to-find errors.

E.g.:

>>> x = [1, 3, 5]

>>> y = x

>>> x[1] = -3

>>> x

[1, -3, 5]

>>> y # !!!!!

[1, -3, 5]

# Make y independent of x

>>> x = [1, 3, 5]

>>> y = x.copy()

>>> x[1] = -3

>>> x

[1, -3, 5]

>>> y

[1, 3, 5]

The basic concept here is that each named object consists of some information such as its class plus a **pointer** to its data. When you make an assignment of one variable to another, the new variable uses the same pointer rather than copying the data. Here are some examples to explore this idea. Note that we can use the “is” operator to see if two variables are using the same data. This is entirely different from “==”.

>>> x = [1, 2, 3]

>>> y = (10, x, 10)

>>> y

(10, [1, 2, 3], 10)

>>> y[1] is x

True

>>> y[0] is y[2]

True

>>> x[1] = 22

>>> y

(10, [1, 22, 3], 10)

>>> del x

>>> y

(10, [1, 22, 3], 10)

When we change some data in “x” we see the change in “y”. But if we delete “x”, then Python sees that both “x” and “y[1]” are pointing to the same data, and it will not delete the data. It only deletes “x” and its pointer to the data. Now “y[1]” is the only pointer to those data. Similarly, if we had re-assigned “x” instead of deleting it, “x” would point to the new data, but “y[1]” would still point to the original data. In other words, the problem only arises when we change the “insides” of a mutable object.

It is worth noting that, depending on the implementation, Python pre-stores some low integers. In my current version, -5 to +256 are pre-stored.

>>> x = 5

>>> y = 5

>>> x is y

True

>>> x = 257

>>> y = 257

>>> x is y

False

>>> x == y

True

***Strong recommendation***: Because Python has only one “None”, “x is None” is good coding, but otherwise you want to check value equality with “==” not “is”.

Here are few more examples:

>>> x = [1, 2]

>>> y = [x, x] # may change via changes to x’s elements

>>> x = (1, 2)

>>> y = [x, x] # safe because x is immutable

>>> y

[(1, 2), (1, 2)]

>>> x = (2, 3)

>>> y

[(1, 2), (1, 2)]

>>> y[0] is y[1]

True

>>> prime\_sq = {2:4, 3:9, 5:25, 7:49, 11:121}

>>> d2 = prime\_sq

>>> prime\_sq[13] = 169

>>> d2 is prime\_sq

True

But, wait! It gets even worse. Even function parameters may be mutable containers, allowing subtle problems:

>>> def sum.valid(y):

for ii in range(len(y)):

if math.isnan(y[ii]):

y[ii] = 0

return(sum(y))

>>> x=[1, 3, math.nan, -4]

>>> sum(x)

nan

>>> sum.valid(x)

0

>>> x

[1, 3, 0, -4]

1. **Indexing and sequence slicing**
   1. After any ***indexable*** container object, you can place square brackets and a single value. Then ***the result no longer has the type of the container***, but of the contents. E.g., [1, "two", 3][1] has the value “two” which is a “str”, not a “list”. And {"a": 1, "b":2}["a"] has the value 1, which is an “int”, not a “dict”. (Like [[]] on a list in R.) An exception is slicing a string; because there is no character non-sequence, the result of indexing with a single value is a string.
   2. Sequence slicing is analogous to aspects of R’s indexing, but practically very different. Objects that are ***sliceable*** (does not include dictionaries or sets) support an expression inside the square brackets containing a “:” which ***returns an object of the*** ***original (container) object type***. (This is similar to using [] as opposed to [[]] on a list in R.)
   3. ***Slicing takes the form of [i:j], or [i:j:s]*** where i, j, and s are integers, and either or both of i and j may be missing. With a single colon, the new object contains all of the original objects with indices that are integers in the range [i, j). If i is missing, zero is used; if j is missing, the length of the object is used. If s is used, it represents a step size. (Slices are similar to [seq(i, j, s)] in R.) Here are some examples:

>>> x = list(range(11))

>>> x

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

>>> x[1] # returns an "int"

1

>>> x[1:2] # returns a "list"

[1]

>>> x[0:10] # length is 10 - 0 = 10

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> x[:]

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

>>> x[:5]

[0, 1, 2, 3, 4]

>>> x[4:6]

[4, 5]

>>> x[8:]

[8, 9, 10]

>>> x[2::3]

[2, 5, 8]

>>> x[::4]

[0, 4, 8]

>>> x[1:5:2]

[1, 3]

* 1. ***Slice indices may also be negative***, which indicates counting backwards from the end:

>>> x[-2:-1]

[9]

>>> x[-3:]

[8, 9, 10

>>> x[:-1]

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

* 1. For mutable sequences, the ***slice notation may be on the left of the equal sign***:

>>> x = [3, 8, 12]

>>> x[0:2] = [1, 2]

>>> x

[1, 2, 12]

>>> x[3:5] = range(2)

>>> x

[1, 2, 12, 0, 1]

>>> x[5:100] = [4, 5, 2] # extra is allowed and dropped

>>> x

[1, 2, 12, 0, 1, 4, 5, 2]

>>> # strangely, pointing past the end only appends values

>>> # and appending ignores the upper value (j).

>>> x[99:100] = range(99, 103)

>>> x

[1, 2, 12, 0, 1, 4, 5, 2, 99, 100, 101, 102]

>>> x = x[:3]

>>> x

[1, 2, 12]

>>> x[5:] = range(5)

>>> x

[1, 2, 12, 0, 1, 2, 3, 4]

>>> x[::2] = range(4) # RHS must be the correct length

>>> x

[0, 2, 1, 0, 2, 2, 3, 4]

>>> x[::2] = [8,9]\*2

>>> x

[8, 2, 9, 0, 8, 2, 9, 4]

1. ***Base functions in Python***
   1. **len()** returns the number of elements in a container, include the number of characters in a “str”. Calling len() on an int, float, complex, bool, None, or a function *is an error*. E.g., len([1, "abc", True, False]) is 4 and len("abc") is 3.
   2. **type()** returns “int”, “complex”, “list”, “tuple”, “str”, “bytes”, etc., and works on anything.
   3. **print()** can be used to show any object(s) with optional controls:

>>> print("x =", 3)

x = 3

>>> print("x=", 3, sep="")

x=3

>>> print("x =", 3, end=", "); print("y = ", 5)

x = 3, y = 5

print() can also output to a file object.

* 1. Like ls() in R, **dir()** returns a list of all objects in your current scope. vars() gives more extensive results, and vars(module\_name) gives module info.
  2. **sorted()** sorts the values and returns them in list form. Add reverse=True to sort in descending order. You can also use key=function\_name to sort on a function of the data, e.g., sorted(x, key=abs), sorts by absolute value.
  3. **input()** allows user input: ans = input("Response? ")
  4. ***The only*** available built-in ***mathematical functions*** are abs(), divmod(), min(), max(), pow(), sum(), and round() (plus % and //):

>>> abs(-3.4)

3.4

>>> divmod(13, 3) # 13 = 4 \* 3 + 1

(4, 1)

>>> min([1, 2, 0, 6])

0

>>> max((1, 2, 0, 6))

6

>>> round(3.4001, 2)

3.4

>>> round(3.4001)

3

* 1. ***Logical functions***

Zero, False, expressions that evaluate to False, empty strings, None, and empty containers are False. Almost anything else evaluates to True.

The any() and all() functions are similar to R.

>>> any([False, 3==1+2, 2>7])

True

>>> all([False, 3==1+2, 2>7])

False

* 1. ***Functions to convert data***:

>>> bin(17) # result is a string

'0b10001'

>>> oct(64+8) # base 8 as a string

'0o110'

>>> hex(255)

'0xff'

>>> hex(946)

'0x3b2'

>>> bool({}) # determine if True or False

False

>>> bool(4.5)

True

>>> complex(3.2)

(3.2+0j)

>>> complex(3.2, 2.3)

(3.2+2.3j)

>>> float("3.4e3")

3400.0

>>> chr(65) # convert code number to character in UTF-8

'A'

>>> ord("A") # convert character to "ordinal" UTF-8 code number

65

>>> ord("\N{greek small letter beta}")

946

>>> "beta: \u03b2"

'beta: β'

>>> str(2.3) # convert any object to a string

'2.3'

>>> frozenset({1, 5, 7}) # makes immutable version

frozenset({1, 5, 7})

>>> list({1,5,7}) # convert various inputs to a list

[1, 5, 7]

>>> list("how now")

['h', 'o', 'w', ' ', 'n', 'o', 'w']

>>> set([3, 2, 3, 1, 2]) # convert various inputs to a set

{1, 2, 3}

>>> tuple([1, 4, 2]) # convert various inputs to a tuple

(1, 4, 2)

>>> tuple({1:'a', 2:'b'}) # or: tuple({1:'a', 2:'b'}.keys())

(1, 2)

>>> tuple({1:'a', 2:'b'}.values())

('a', 'b')

* 1. See <https://docs.python.org/3/library/functions.html> for a full list of built-in functions.
  2. Use help(myFunction) to learn about a function.

1. Intro to **object oriented** aspects of Python
   1. Each value in Python is of a particular ***class,*** checked with type() vs. class() in R. Each class can be un-instantiated or can have one or more **instantiations**. E.g., the math module (see below) is un-instantiated because although it contains many useful math functions, you cannot create any objects whose class (type) is “math”. In contrast, a “customer” class might have one instantiation (variable or element in a collection) for each customer.
   2. Each class may have **class variable(s)** shared by all instances of the class as well as **instance variable(s)**, which are data unique to each instance (variable). In addition, each class has a set of **method functions** designed to work independently or with the instances. E.g., a customer class object may have instance variables for the customer’s status (gold vs. silver vs. platinum), age, current balance, etc. There would be method functions to retrieve and update these values, and possibly one to compute some aggregate customer value using the values of several of the instance variables.
   3. It is a first principle of Computer Science that ***users of a class need not understand the details of implementation***, and the implementation details may change in the future with no untoward downstream effects because users should generally be relying on ***access methods*** rather than class variables (except for ones that are fundamentally unchangeable).
   4. In addition to ***help***(class\_name) and help(object\_name), Spyder lets you type “Tab” after an object name and a period to see the method and variable choices.
   5. Python uses the standard ***period notation*** to refer to the methods and variables of a class. Depending on context, the period may follow the class name or the name of an instantiated class object. It may be difficult to know if a given choice is a method or a variable, i.e., whether to use parentheses (and possibly arguments) or not. (Some built-in functions seem like they would have been better implemented as methods.)
   6. Technical note: There are several ***complaints about class implementation in Python***. One is that there is no real way to make private variables (there is a convention of starting them with two underscores). A related one is that users can alter the data in a way that makes the methods fail. Another is explicit passing of “self”.
   7. A list is a good ***example of a class***. Methods include append(), clear(), copy(), count(), extend(), index(), insert(), pop(), remove(), reverse(), and sort().
      1. append(), clear(), extend(), insert(), pop(), remove(), reverse(), and sort() are all “destructive”, changing the object. All return nothing (None) except pop() which removes the last value and returns what was removed.

>>> x = [1, 2, 0, 5]

>>> x.append(3)

>>> x

[1, 2, 0, 5, 3]

>>> x.append([6, 7])

>>> x

[1, 2, 0, 5, 3, [6, 7]]

>>> len(x)

6

>>> print(x.pop())

[6, 7]

>>> x

[1, 2, 0, 5, 3]

>>> x.extend([6, 7])

>>> x

[1, 2, 0, 5, 3, 6, 7]

>>> x.sort()

[0, 1, 2, 3, 5, 6, 7]

>>> x.reverse()

[7, 6, 5, 3, 2, 1, 0]

>>> x.remove(2) # error if not in list

>>> x

[7, 6, 5, 3, 1, 0]

>>> x.clear()

>>> x

[]

* + 1. copy(), count(), and index() leave the list unchanged

>>> x = [5, 0, 2, 7, 5]

>>> y = x.copy()

>>> y[0] = -1

>>> x

[5, 0, 2, 7, 5]

>>> x.count(5)

2

>>> x.index(5)

0

>>> x[1:].index(5)

3

* + 1. The only variable in the class is the private variable “\_\_hash\_\_”.

type(x.\_\_hash\_\_)

NoneType

* + 1. Another example is the str class for strings. Because they are immutable, they cannot be changed in place, but you can use a method that returns an altered string and assign that to the same or a different variable: x = x.upper().
    2. See future handout for defining your own classes

1. **Comprehensions** are a powerful language feature that allows generation of useful lists and dictionaries. They are similar to the apply() functions in R.
   1. **List comprehensions** have the basic syntax [expression\_with\_var for var in values]. The result is a list.

>>> [x\*\*2 for x in range(1,21,2)]

[1, 9, 25, 49, 81, 121, 169, 225, 289, 361]

>>> [(x, x\*\*2) for x in range(5)]

[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16)]

* 1. **Dictionary comprehensions** work similarly:

>>> {k: chr(k+ord("A")) for k in range(26)}

{0: 'A', 1: 'B', 2: 'C', 3: 'D', 4: 'E', 5: 'F', 6: 'G', 7: 'H',

8: 'I', 9: 'J', 10: 'K', 11: 'L', 12: 'M', 13: 'N', 14: 'O',

15: 'P', 16: 'Q', 17: 'R', 18: 'S', 19: 'T', 20: 'U', 21: 'V',

22: 'W', 23: 'X', 24: 'Y', 25: 'Z'}

* 1. The built-in function **zip()** takes ***multiple vector arguments***, reduces to the shortest length, then “serves up” tuples of one value from each argument.

>>> [a + b for (a,b) in zip(["car", "hip", "stat-"],

["toon", "ster", "wizard"])]

['cartoon', 'hipster', 'stat-wizard']

>>> x1 = [2, 5, 7]; x2 = [1, 4, -2]

>>> [5 + 3\*x1 + 2\*x2 for (x1, x2) in zip(x1, x2)]

[13, 28, 22]

* 1. Comprehensions may have an ***if clause***:

>>> [x\*\*2 for x in range(20) if x % 2 == 1]

[1, 9, 25, 49, 81, 121, 169, 225, 289, 361]

>>> [x\*\*2 for x in range(1,21) if x % 3 == 0]

[9, 36, 81, 144, 225, 324]

* 1. ***Multiple for clauses are allowed*** (first moves slowest)

>>> [(y, y - x) for x in (3, 2, 1) for y in (5, 10)]

[(5, 2), (10, 7), (5, 3), (10, 8), (5, 4), (10, 9)]

>>> [(x, y, x+y) for x in range(5) for y in range(3)]

[(0, 0, 0), (0, 1, 1), (0, 2, 2),

(1, 0, 1), (1, 1, 2), (1, 2, 3),

(2, 0, 2), (2, 1, 3), (2, 2, 4),

(3, 0, 3), (3, 1, 4), (3, 2, 5),

(4, 0, 4), (4, 1, 5), (4, 2, 6)]

1. **Modules**
   1. ***Additional functionality*** beyond standard Python is often required and available in modules either in the Standard Python Library, written by others but not in the S.P.L., or written by you.
   2. For example, the “math” module defines the math class, with the following methods (functions) ceil(), floor(), factorial(), isclose(), isfinite(), isinf(), isnan(), trunc(), exp(), log(), log1p(), log2(), log10(), sqrt(), sin(), cos(), tan(), asin(), acos(), atan(), degrees(), radians(), gamma(), and some others. There are also some class constants: pi, e, inf, and nan.
   3. Often the best form is:

import math

f = math.floor(y)

* 1. A good alternative is (which only adds the needed functions):

from math import ceil, floor

f = floor(y)

* 1. Another form is:

import math as m

f = m.floor(y)

* 1. Least desirable (because it hides dependencies) is

from math import \*

f = floor(y)

* 1. See <https://docs.python.org/3/library/> for standard library modules. Commonly used ones include string (additional string functions and constants), re (regular expressions), datetime, collections, array, cmath (complex math), random, statistics, itertools (for functional programming), os.path, tempfile, shutil (copy, rename, delete files), zipfile, csv, logging, getpass, threading, email, mimetypes, html, webbrowser, audioop, wave, locale, tkinter, etc.

1. ***Special values***
   1. Unlike R, ***Python does not have “NA”****.*
   2. Some ***limits may be implementation dependent***. If you want/need to see/use them, use:

>>> import sys

>>> sys.float\_info

sys.float\_info(max=1.7976931348623157e+308, max\_exp=1024, max\_10\_exp=308, min=2.2250738585072014e-308, min\_exp=-1021, min\_10\_exp=-307, dig=15, mant\_dig=53, epsilon=2.220446049250313e-16, radix=2, rounds=1)

* 1. Python has ***constants for infinity and minus infinity***. Note that expressions or values that silently store Inf in R, will instead cause an exception in Python. If you want to store the infinity special value when a “float division by zero” or “math domain error” is thrown, you will need to handle the exception in a try block. Although float('inf') is a weird valid way to generate inf in Python, to be more consistent with standard practice, you should use:

>>> import math

>>> infinity = math.inf

>>> infinity

inf

>>> minus\_infinity = - math.inf

>>> math.isfinite(minus\_infinity)

False

>>> math.isinf(minus\_infinity)

True

* 1. Python has a ***constant for “not a number”*** which shows as “nan”. Again, expressions or values that silently store NaN in R, will instead cause an exception in Python. You will normally use “nan” in Python to represent both “not a number” and “missing value”. Again, although float(‘nan’) is valid, it is better to use the built-in constant:

>>> import math

>>> bad\_value = math.nan

>>> bad\_value

nan

>>> math.isfinite(bad\_value)

False

>>> math.isinf(bad\_value)

False

>>> math.isnan(bad\_value)

True

* 1. math.pi is the maximally precise IEEE floating point ***representation of .***

1. **A generator** is a particular type of **iterator**, and iterators are objects that create values that can be used one-by-one without necessarily storing them all at once. They are the “guts” of comprehensions and the range() function. You can think of them as little programs that generate the values, rather than, say, an actual list of the values. E.g., range(1000000) is a small object, while list(range(1000000)) is a large object. Generators are functions that use the yield statement to save their state and return one value. For now, we will just use **a generator expression**, which writes the generator for us. Syntactically, a generator expression looks the same as the inside of a comprehension. If it is going to be directly assigned to a variable name, it must be placed inside of parentheses. If it is the argument to set(), tuple(), or list(), the parentheses are not needed. A generator object may be used as the source of values in a for statement. When a generator is used, it is used up (see below)! Here is an example:

import math

gen = (v\*\*2 for v in range(10))

t = tuple(gen)

print(len(t)) # 10

u = tuple(gen)

print(len(u)) # 0

gen = (v\*\*2 for v in range(10))

for v2 in gen:

print(math.sqrt(v2))

If you print gen, the result is something like “<generator object <genexpr> at 0x0000000009A0B1A8>”. If you print t, you get a tuple with 10 values (squares). The length of u is 0, because the generator was “used up” and has no more values to generate. The final three lines print “0.0”, …, “9.0”.

Perhaps the most important thing to remember about generators is that t = tuple(v\*\*2 for v in range(10)) is ***more efficient*** in time and space usage than t = tuple([v\*\*2 for v in range(10)]) because the list comprehension in the second version actually allocates and stores a list and then copies it to the tuple, while the first version, using the generator, only creates the tuple.

1. ***Writing functions***
   1. The ***style*** guide suggests two blank lines before a function definition, then the initial “def” line (name and arguments), then a triple quoted **documentation string**, then the indented function code. Use the return statement to return a value of any kind.
   2. Example 1:

def add\_two\_lists\_and\_constant(list1, list2=None, constant=0):

""" Add two lists and a constant

Positional argument:

list1 -- a list of numbers (int, float, or complex)

Keyword arguments:

list2 -- a second list of numbers of the same size (or None)

constant -- a numeric constant

"""

if not isinstance(list1, list):

raise TypeError("'list1' should be a list")

if list2 is not None and not isinstance(list2, list):

raise TypeError("'list2' should be a list or 'None'")

if list2 is not None and len(list2) != len(list1):

raise IndexError("the lists should be the same length")

if (not isinstance(constant, (int, float, complex))):

raise TypeError("the constant should be numeric")

if list2 is None:

return [x + constant for x in list1]

return [x + y + constant for (x, y) in zip(list1, list2)]

# Tests that fail

add\_two\_lists\_and\_constant(7)

add\_two\_lists\_and\_constant([7, 2], 7)

add\_two\_lists\_and\_constant([7, 2], [1])

add\_two\_lists\_and\_constant([7, 2], [1, 2], "1")

# Tests that succeed

add\_two\_lists\_and\_constant([7, 2], [1, 2])

add\_two\_lists\_and\_constant([7, 2], [1, 2], -8)

Comments on new syntax:

* “def”, “if”, “elif”, and “else” lines end in “:”
* The style guide says to indent 4 without using Tab.
* isinstance() takes an object and an object type and works like R’s is().
* “list”, “int”, “float”, and “complex” are built-in “object types”
* Checking for None is done with is which checks identity rather than type.
* raise generates an exception (that can be caught)
* See <https://docs.python.org/3/library/exceptions.html#bltin-exceptions> for built-in exceptions or use Exception() to generate your own.
* We could add code to check that all data is numeric.
  1. Example 2:

def stem\_and\_leaf(data, interval=1):

""" Stem-and-leaf plot

data -- a tuple or list with numeric values

interval -- a number defining the bin width

The plot is printed as a side effect.

The function returns None.

"""

from math import ceil, floor

if not isinstance(data, (tuple, list)):

raise TypeError("'data' must be tuple or list")

first\_interval, maxData = min(data), max(data)

first\_interval = interval \* floor(first\_interval / interval)

last\_interval = interval \* ceil(maxData / interval)

if last\_interval == maxData:

last\_interval += interval

interval\_count = int((last\_interval-first\_interval)/interval)

indices = range(interval\_count)

starts = [first\_interval + index\*interval

for index in indices]

counts = [len([1 for d in data

if ii <= d < ii + interval])

for ii in starts]

for ii in indices:

print(str(starts[ii]) + '\t' + "\*"\*counts[ii])

return None

stem\_and\_leaf([10, 14, 13.5, 12, 13.4, 18.5], interval=2)

stem\_and\_leaf((0.12, 0.15, 0.18, 0.25, 0.32, 0.59), 0.2)

Comments on new syntax:

* +=, -=, \*=, and /= may make for cleaner (easier to read) code
* “for” loops are similar to R in that they iterate over a “vector” (or generator). The loop ends at the first “dedented” line. The “break” and “continue” statements are available as in R. Parentheses are not used.
* “while” loops are also available.
* A better version of this function would format the interval starts.
  1. **Variable scope**
     1. “scope” defines where in a program a particular variable is accessible.
     2. A good reference is <http://python-textbok.readthedocs.io/en/1.0/Variables_and_Scope.html>
     3. “A variable which is defined in the main body of a file is called a **global variable**. It will be visible throughout the file, and also inside any file which imports that file. Global variables can have unintended consequences because of their wide-ranging effects – that is why we should almost never use them. Only objects which are intended to be used globally, like functions and classes, should be put in the global namespace.”
     4. “A variable which is defined inside a function is **local** to that function. It is accessible from the point at which it is defined until the end of the function, and exists for as long as the function is executing. The parameter names in the function definition behave like local variables, but they contain the values that we pass into the function when we call it. When we use the assignment operator (=) inside a function, its default behaviour is to create a new local variable – unless a variable with the same name is already defined in the local scope.”
     5. As opposed to R, a variable in a Python function is either entirely local or entirely global. Consider this R example:

x = 5

f = function() {

print(x)

x = 3

return(x)

}

print(f())

print(x)

This first prints “5” from the global x via the print() inside of f(). Then it prints “3” which is the value of the returned local variable “x”. Finally it prints 5 because after f() is run, it’s local variables go out of scope and the global value of “x” is the only one available to the final print().

In contrast, in Python,

x = 5

def f():

print(x)

x = 3

return x

print(f())

print(x)

gives “UnboundLocalError: local variable 'x' referenced before assignment”. But if we remove the first print(), it works fine and prints the local “3” from inside the function, then the global “5” for the last line.

* + 1. Python has a **global statement.** If we precede the first “print(x)” above with “global x”, then “x” has global scope throughout the function, and the final “print()” prints “3” because the function dangerously changes the global value of “x”. Use of a “global” statement is a good thing if you decide you need to refer to a value defined before the function is defined because it make that fact clear, but you should almost never assign to a global variable inside a function.

1. ***Debugging in Spyder*** (using iPython debugger)
   1. Basic strategy:
      1. In the editor, open a file with your code.
      2. Double click on the left to define one or more **breakpoint**(s).
      3. Click “Debug File”. Click “Continue execution until next breakpoint”.
      4. Examine the Variable Explorer to detect any problems. Type expressions in the IPython console (at the “ipdb >” prompt). Some expressions such as "list(x)" conflict with the small number of debugger commands; use "!list(x)" instead.
      5. Click “Run Current Line” to execute one more line. If the current line calls another one of your functions and you want to step through that one, use “step into function or method of current line”. If you want to complete the “detour” and return to the main code, click “run until current function or method returns”. Repeat as needed, progressing through your code.
      6. If you are ready to fix your code, click “stop debugging”.